Journal of Web Engineering, Vol. 11, No.2 (2012) 119-130

A CACHING MECHANISM FOR QOS-AWARE SERVICE COMPOSITION

QUANWANG WU
Chongqing University, Chongqing, China
wqw@cqu.edu.cn

QINGSHENG ZHU
Chongqing University, Chongqing, China
qszhu@cqu.edu.cn

PENG LI
Chongqing University, Chongqing, China
cheney@cqu.edu.cn

Received July 29, 2011
Revised January 8, 2012

Web service composition enables seamless and dynamic integration of business applications on the web. With the growing number of web services that provide the same functionality but differ in quality parameters, the QoS-aware service composition becomes a decision problem on which component services should be selected such that the quality of the composite service is optimized and user preference is satisfied. In this paper, we presented a caching mechanism for this problem, which can be complementary to most of current approaches to enhance the efficiency. We evaluate our approach experimentally using a real QoS dataset and it shows a significant impact in reducing the computing time.

Key words: Service Composition, QoS, Caching Mechanism, Hit Ratio, Genetic Algorithm
Communicated by: M. Gaedke & T. Tokuda

1 Introduction

Recently, the trend for businesses to outsource parts of their processes, in order to focus on their core activities has becoming more and more popular. At the same time, users often need to compose different services to achieve a more sophisticated task that cannot be fulfilled by an individual one. According to the Service Oriented Architecture paradigm, composite applications are specified as abstract processes composed of a set of abstract services. Then, at running time, each abstract service is instantiated by a concrete web service [9]. This ensures loose coupling and flexibility of the design.

Quality of Service (QoS) is crucial for service providers to meet the requirements of customers. QoS parameters include generic ones such as response time, availability, reputation, as well as domain-specific ones like throughput for multimedia web services. Since users or intelligent agents

have diverse preferences for QoS parameters, e.g. Tom can bear a composite service with a response time of 2 seconds at most or he deems availability much important than service price, QoS-aware service composition aims at finding a composite service with the optimal aggregated QoS based on the specific preference. This problem can be modelled as a multi-dimension multi-choice 0-1 knapsack problem, which is known to be NP-hard, i.e. it is expected that any exact solution to this problem has an exponential computational complexity [15].

With the increasing number of functionality-equivalent services offered on the web at different QoS levels, QoS-aware service composition becomes especially significant and challenging. There has been a more than 130% growth in the number of published web services in the period from October 2006 to October 2007 [1]. The statistics published by the web service search engine Seekda (http://webservices.seekda.com/) also indicate an exponential increase in the number of web services over the last three years. Moreover, with the proliferation of the Cloud Computing and Software as a Service (SaaS) concepts, more and more web services will be offered on the web. Thus, it is expected that service requesters will be soon faced with a huge number of functionality-equivalent services offered at different QoS levels, and an efficient approach for an automatic service selection and composition is quite on demand.

In this paper, we present a technique to improve the efficiency by adopting the caching mechanism for QoS-aware service composition. This choice is motivated by three reasons. First, one user can request a composite service with the same preference for more than one times. Second, there must be some similarities between different requesters' preferences. Finally, the computational complexity for cache searching to acquire a concrete composite service is much lower than other approaches [6, 15, 17]. Each time the QoS-aware composition engine instantiates an abstract composite service based on a specific preference, the preference with the corresponding solution (i.e. the concrete composite service) will be cached for that composite service. Then, when a new request for that composite service comes, the engine will first look up the cache for the solution. This caching mechanism can be complementary to most of current approaches [6, 15, 17] for QoS-aware service composition to enhance the efficiency.

The main contributions of this paper are two-fold. First, a caching mechanism for the problem of QoS-aware service composition is presented and it is applied on the genetic algorithm. Secondly, an exploratory trial is conducted to discretize the continuous QoS constraints and QoS priorities in order to boost the hit ratio of the caching mechanism, and the empirical study is shown.

The remainder of this paper is organized as follows: Section 2 discusses the related work. Section 3 describes the problem of QoS-aware service composition, while Section 4 presents an approach based on genetic algorithm. Our caching mechanism is presented in Section 5. Empirical studies are shown in Section 6. Finally, Section 7 concludes and gives future work.

2 Related Work

Many researches have been carried out on QoS-aware service composition.

In [17], Zeng et al. present an extensible QoS computational model and propose two approaches to address the problem: local optimization and global optimization. The former one is quite efficient, but it can not guarantee global optimality and can not support global QoS parameter constraints. The latter one uses mixed linear programming technique to find the globally-optimal service components for the composition. Linear programming is very effective when the size of the problem is small, but suffers from poor scalability because of the exponential time complexity.

Due to the poor scalability of linear programming, more efficient heuristic algorithms for the problem are proposed to find a near-to-optimal solution.

In [6], Gerardo et al. propose an approach based on genetic algorithm, where the composite service is encoded as chromosome, and the fitness is the aggregated QoS of the composite service. The algorithm starts with a random set of chromosomes and evolves through chromosome selection, mutation and crossover. The vision is that the fitness will increase from generation to generation and the fittest chromosome represents the (nearly-) optimal solution of the problem.

In [15], Yu and Lin propose two models for this issue: a combinatorial model (WS HEU) and a graph model (MCSP-K). A heuristic algorithm is introduced for each model. The time complexity of the heuristic algorithm for the combinatorial model is polynomial, whereas the complexity of the heuristic algorithm for the graph model is exponential.

In [14], Ye and Mounla present a service composition scheme that uses a combination of linear programming, case-based reasoning, The scheme reduces the computing time of service composition by reusing existing compositions and the roulette wheel selection is used for selecting an existing solution to cope with the fluctuation of QoS. However, global QoS constrains are not considered in this scheme.

The proposed skyline based algorithm can be used as a pre-processing step to prune non-interesting candidate services and hence reduce the computing time of the applied composition algorithm [4].

Cache is a component that transparently stores data so that future requests for that data can be served faster, and it is widely applied in memory architecture and web browsers. In [12], a caching mechanism for semantic web service discovery is presented, which exploits structural knowledge and previous discovery results for reducing the search space of consequent discovery operations.

The caching mechanism presented in this paper also aims to complement other QoS-aware service composition algorithms like [4] and its principle is similar to the case-based reasoning mechanism in [14] to some extent.

3 QoS-aware service composition model

3.1 Abstract & Concrete Composite Service

In our model we assume that we have a universe of web services $\mathbb{S}$ which is defined as a union of abstract service classes. Each abstract service class $S_j \in \mathbb{S}$ is used to describe a set of functionally-equivalent web services.

An abstract composite service is defined as a composition of abstract service classes: $CS_{abstract} = \{S_1, S_2, \ldots, S_n\}$, and n is the number of abstract service classes in $CS_{abstract}$. It can be stated in a workflow-like language (e.g. BPEL [10]). A concrete composite service is defined as an instantiation of an abstract composite service. This can be obtained by binding each abstract service class in $CS_{abstract}$ to a concrete web service. We use CS to denote a concrete composite service.

Figure 1 gives a better idea of the QoS-aware service composition problem. Given a $CS_{abstract}$, the discovery engine uses a service registry (e.g. UDDI) to locate available web services for each abstract service class in $CS_{abstract}$ using syntactic or semantic functional matching through service descriptions. Thus, a list of candidate services is obtained for each abstract service class. The goal of QoS-aware service composition is to select one concrete service from each list to form a CS, whose aggregated QoS values are optimal while user's end-to-end QoS preferences are also satisfied.

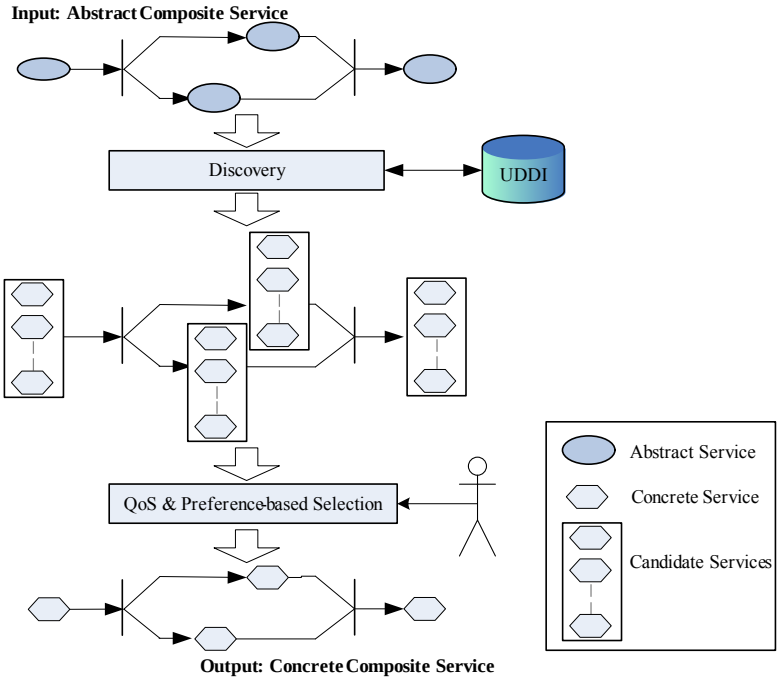

Figure 1 QoS-aware service composition

3.2 QoS Criteria and QoS Aggregation

QoS parameters can include generic QoS parameters like response time, availability, reputation etc, as well as domain-specific QoS parameters like throughput for multimedia web services. We use the vector $Q_s = \{q_1, q_2, \ldots, q_r\}$ to denote the QoS parameters of service s, where q_i denotes the i-th quality parameter and r represents the number of concerned QoS parameters.

Aggregation type	**Examples**	**Function**
Summation	Response time, Price	$q(CS) = \sum_{j=1}^{n} q(s_j)$
	Reputation	$q(CS) = 1/n \sum_{j=1}^{n} q(s_j)$
Multiplication	Availability, Reliability	$q(CS) = \prod_{j=1}^{n} q(s_j)$
Minimum	Throughput	$q(CS) = \min_{j=1}^{n} q(s_j)$

Table 1 Examples of QoS aggregation functions

The set of QoS parameters can be divided into two subsets: positive and negative. The values of positive parameters need to be maximized (e.g. reputation and availability), whereas the values of negative parameters need to be minimized (e.g. price and response time). For the sake of simplicity, in

this paper we consider only negative ones (the positive ones can be easily transformed into negative ones by multiplying their values by -1).

The QoS value of a composite service is decided by the QoS values of its component services as well as the composition model used (e.g. sequential, parallel, conditional and/or loops). We focus on the sequential composition model, since other models may be reduced or transformed to the sequential model [7]. In our model we consider three types of QoS aggregation functions: 1) summation, 2) multiplication and 3) minimum relation. Table 1 shows examples of these aggregation functions.

3.3 Problem statement

In order to evaluate the multi-dimensional quality of a given web service, a utility function is used. In this paper we use the Simple Additive Weighting (SAW) [16] as the utility function to map the quality vector Q_s into a single real value so as to enable sorting and ranking of service candidates.

There are two phases in applying SAW: scaling and weighting. The former normalizes the QoS parameters' values into a value between 0 and 1, by comparing it with the minimum and maximum possible value. The scaling process is then followed by a weighting process which represents user priorities on different QoS parameters.

For a concrete composite service $CS = \{s_1, \ldots, s_n\}$, the aggregated QoS values are compared with minimum and maximum possible aggregated values in the $CS_{abstract}$ it belongs to. The minimum (or maximum) possible aggregated values can be easily estimated by aggregating the minimum (or maximum) value of each service class in the $CS_{abstract}$. For example, the maximum execution price of *CS* can be computed by summing up the execution price of the most expensive service candidate in each service class. The overall utility function of a composite service is computed as

$$U(CS) = \sum_{i=1}^{r} \frac{Q_{\max}(i) - q_i(CS)}{Q_{\max}(i) - Q_{\min}(i)} . w_i \tag{1}$$

where, $q_i(CS)$ represents the aggregated q_i value of *CS*, $Q_{min}(i)$ and $Q_{max}(i)$ denote the minimum and maximum possible aggregated values, respectively, and $w_i \in [0,1]$ is the weight of q_i to represent user priorities with $\sum_{i=1}^{r} w_i = 1$.

Global QoS constraints represent user's end-to-end QoS requirements. As mentioned earlier, we only consider negative QoS criteria so in our model we only have upper bound constraints. The constraint vector $\boldsymbol{C}$ is denoted as $\{c_1, \ldots, c_r\}$, and c_i is the constraint for $q_i(CS)$. In addition, the case of $c_i = \infty$ indicates no constraint for $q_i(CS)$.

Hence, the problem of QoS-aware service composition is to seek a *CS* for $CS_{abstract}$ according to a specific user's preference including priority weights and constraints for QoS, where

1. The overall utility $U(CS)$ is maximized, and
2. The aggregated QoS satisfy: $\forall\, c_i \in \boldsymbol{C}$, $q_i(CS) \leqslant c_i$.

4 QoS-aware Service Composition based on Genetic Algorithm

Without limits on the linearity of the QoS composition operators (e.g. utility function and constraints), genetic algorithm (GA) is the most promising and generic heuristic for QoS-aware service composition. Thus, we employ GA as our basic method and then equip it with the caching mechanism.

In this case, a composite service is encoded as a chromosome, and the chromosome is represented by an integer array with its length equal to the number of component services. Each item in the array,

in turn, contains an index to the list of the candidate services matching that abstract service. The crossover operator is the standard two-point crossover, while the mutation operator randomly selects a component service (i.e., a position in the chromosome) and randomly replaces the selected candidate service with another one in the list.

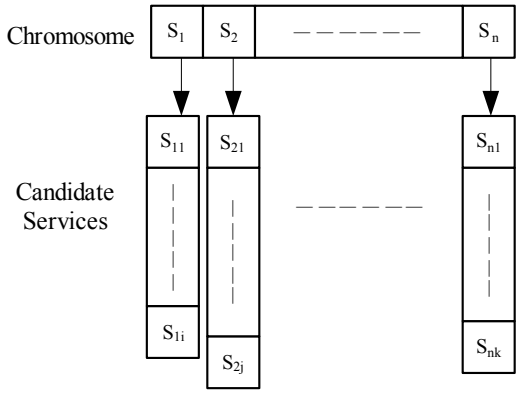

Figure 2 Encoding of Chromosome

We modify the utility function *U(CS)* in section 3.3 to encompass punishment for those concrete composite services that do not meet the constraints as follows:

$$U'(CS) = \sum_{i=1}^{r} q_i^{'}(CS).w_i \tag{2}$$

where:

$$q_i^{'}(CS) = \begin{cases} -1 & if \;\; c_i \neq \infty \;\; and \;\; q_i(CS) > con_i \\ \dfrac{Q_{\max}(i) - q_i(CS)}{Q_{\max}(i) - Q_{\min}(i)} & if \;\; c_i = \infty \;\; or \;\; q_i(CS) \leq con_i \end{cases} \tag{3}$$

We use *U'(CS)* as the fitness function for the genetic algorithm. Finally, the stop criterion for the GA is defined as follows:

1. Set a large number as a maximum number of iterations for the GA.

2. Set a fixed number of generations: $maxgen_{fitness}$. If the chromosome with the best fitness remains unchanged for $maxgen_{fitness}$ generations, GA will abort and return the concrete composite service with highest fitness value.

3. If the chromosome with the highest fitness value does not satisfy the constraints, the next highest one which satisfies will be returned. If no chromosome satisfies, then we judge no solution has been found.

5 A Caching Mechanism for QoS-aware Service Composition

To solve QoS-aware service composition, Linear Programming needs an exponential time complexity, and heuristic algorithms also require a large sum of computing time when the size of the problem is large. However, no matter how many component services and service candidates there are, the time complexity of looking up a cache pool remains only *O(Capacity of Cache)* all the time.

Since one user can request a composite service with the same preference for more than one time and there must be some similarities between different users' preferences, we determine to employ a caching mechanism to reduce computing time for the problem of QoS-aware service composition.

For the sake of clarity and focus, we set a cache for each composite service respectively instead of a cache for all composite services. This is also because every composite service may have its own particular characteristic so that we could set different cache parameters respectively.

5.1 Cache Entry

Generally, a cache is made up of a pool of entries and each entry consists of a tag and a datum. Cache capacity represents the maximal number of entries the cache pool can have. The problem of QoS-aware service composition is to seek a best *CS* for $CS_{abstract}$ according to a specific user preference (*Pref*), so the cache entry can be denoted as <*Pref*, *CS*>, where the former element represents the tag, and the latter is the datum.

We also append the time stamp (t_s) to the cache entry, which indicates when it is stored. Validity time of cache (t_v) is set to restrain the obsolete cache entries which are not consistent with the current situation. The cache entry is valid if t_v plus t_s is greater than the current time. In addition, we append the utility value *U(CS)* of the composite service to the entry for comparison with the real QoS performance. Thus, the cache entry is represented as <*Pref*, *CS*, t_s, *U(CS)*>.

Assuming an integer requires 4B memory size, time stamp and float value require 8B (e.g. in Java), and four QoS parameters are considered, then, for a composite service who has 50 component services, the memory space each element in the cache entry requires is:

Pref	(8B+4B)*|QoS| = 48B
CS	4B * 50 = 200B
t_s	8B
U(CS)	8B

Table 2 Memory space for each element in the cache entry

The cache entry requires a memory space of 264B altogether and the cache pool with a capacity of 10,000 requires almost 2.5MB memory, which can not be neglected in the system design. Thus, we need to utilize caching mechanism for composite services which have numerous service candidates or service components, or which are used pretty frequently. On the contrary, it will be not so worthwhile to utilize the caching mechanism.

5.2 Caching mechanism

Each time the QoS-aware composition engine instantiates a $CS_{abstract}$ based on a specific user preference, the entry < *Pref*, *CS*, t_s, *U(CS)*> will be cached for the $CS_{abstract}$. Then, when a new request for this composite service comes, the engine will first look up the cache for a solution. If an entry can be found with a matching tag and a valid time stamp, i.e. $Pref = Pref_{request}$ *and* $t_v + t_s \geqslant time_{current}$, the datum in the entry is used. This situation is known as cache hit.

In the case of cache hit, the real QoS data are monitored during the execution of *CS* and the real overall utility value of *CS* is computed in order to compare with the predicted value *U(CS)*. If the absolute value of the difference between the two values is greater than a threshold θ (Eq.4), which indicates the network environment has changed and the used cache entry is obsolete, the specific cache entry will be removed.

$$\frac{\left|u_p - u_a\right|}{u_p} \geq \theta \tag{4}$$

where, u_p denotes the predicted value *U(CS)* and u_a denotes the actual overall utility value.

In the alternative case known as cache miss, where the cache is consulted and found not to contain a datum with the desired tag, the composition engine will take the basic algorithm to achieve a solution of *CS*, and then cache this new solution.

When the cache pool is full, the new-coming entry should replace an old entry. The heuristic used to select the entry to eject is known as the replacement policy. Popular replacement policies include First Come First Out (FCFO), Least Recently Used (LRU), Random (RAND). Their performance in this context will be compared in the experiment section.

In order to adapt to the dynamic environment, we set a self-adaptation strategy for validity time t_v. In the case of cache hit, if Eq.4 is true, which indicates that the cache is effective, t_v will increase by a small value. Otherwise, t_v will decrease by a larger value. In addition, t_v can also be limited by an upper and lower bounds. In a nutshell, t_v should be set large in a less dynamic network and be set small in a highly dynamic network.

5.3 Discretization of User Preference

In the QoS-aware service composition problem, preferences including QoS parameter constraints and priorities are traditionally continuous, which can render hit ratio of cache extremely low. In order to enhance the hit ratio, we lead an exploratory trial to discretize the continuous QoS constraints and QoS priorities.

For each QoS parameter, we set five priority levels for users to choose: Lowest, Low, Medium, High and Highest. These levels are mapped to integral numbers "1," "2," "3," "4," and "5," respectively. Priority weights can be obtained based on user's selections of priority levels. For example, given that the vector of user's selections is $\{p_1, \ldots p_r\}$, where p_i represents the priority level of the i-th QoS parameter, the i-th priority weight w_i is calculated as:

$$w_i = p_i / \sum_{i=1}^{r} p_i \tag{5}$$

As for the QoS constraints, we can not directly discretize them like QoS priorities, which can lose generality. From a different perspective, we list a number of representative constraint values for users to choose. After running for a period of time, the QoS-aware composition engine can learn the percentage of users who have set constraints on each QoS parameter, based on the previous user requests. Moreover, for each QoS parameter, the dataset of constraint values are grouped to a certain number of clusters by clustering algorithm (e.g. K-means) and the representative constraint values are set as the centroids of these clusters. Hence, the list of representative constraints is recommended for users to set QoS constraints. Meanwhile, users can also set constraint values that are not in the list but it is not encouraged. Representative constraint values are reselected regularly by clustering algorithm.

6 Empirical Study

6.1 Assumptions

To the best of our knowledge, there is no available dataset of user preferences on QoS. Thus, for the further analysis, we assume the probability for consumers to set a constraint on a QoS parameter to be p_c. The values of representative constraints for each QoS are set empirically as:

$$c_i = Q_{\min}(i) + (Q_{\max}(i) - Q_{\min}(i)) * k / n_c,\ 1 \le \mathrm{k} \le \mathrm{n_c} \quad (6)$$

where n_c is the total number of representatives. We set n_c=10 in the following evaluations.

Let *P(Level)* be the probability that a user's priority level is chosen for a specific QoS parameter. We assume that the distribution is symmetric around the neutral priority level (Medium), i.e. *P(Low)=P(High)* and *P(Lowest)=P(Highest)*, and we also assume that the ratio between *P(Medium)* and *P(High)* is equal to the ratio between *P(High)* and *P(Highest)* , denoted as λ.

6.2 Study 1: hit ratio

We first analyze hit ratios under different values of p_c and λ. The cache pool with a capacity of 2000 is assumed to be already full, the utilized replacement policy is LRU, and four QoS parameters are considered. We simulate 100,000 new user requests in each case of p_c and λ. Fig. 3 illustrates the hit ratio of the caching mechanism. According to this figure, we can learn that with the growing of p_c, the hit ratio drops drastically. The hit ratio is the lowest when λ=1 (that is, the probabilities of all priority levels are equivalent) and then, the hit ratio gradually rises with the growing of λ. In a nutshell, the trend is that smaller p_c values can lead to higher hit ratios while larger λ values (λ>1) also lead to higher ratios.

Afterwards, we check hit ratios under different cache capacities. Cache capacity has a large impact on the performance of cache. If it is set too small, the hit ratio will be low. On the other side, the space and time complexity will be augmented and it also leads to obsolete entries which do not correspond with the current network. In this experiment, we set p_c to 0.3 and λ to 2. In each case of the capacity, 100,000 user requests are simulated and we apply FCFO, LRU and RND as the replacement policies, respectively.

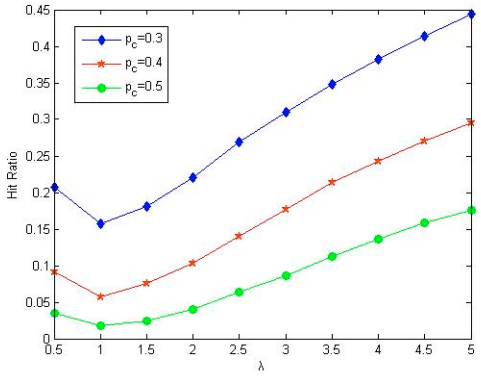

Figure 3 Study 1 with different λ and p_c values

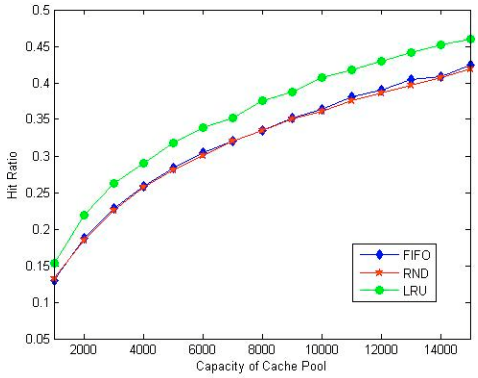

Figure 4 Study 1 with different capacity values

Figure 4 depicts the relationship between hit ratio and capacity, and the comparison of different replacement policies. With the capacity increasing, the hit ratio augments quickly at the initial time and then the acceleration gradually decreases. Among the three replacement policies, the performance of FIFO and RND are nearly the same, and LRU gain the best performance since it always keeps cache entries with user preferences of higher probability.

As the two simulations in this section have not considered the case that the same consumer can request more than one times with the same preference, the hit ratio in the real application can be higher than the aforementioned results to some extent.

6.3 Study 2: cached GA

We equip the genetic algorithm in section 4 with the caching mechanism (called Cached GA) and use the common GA as the benchmark. The detailed characteristics for GA are as follows: the best two individuals kept alive across generations (i.e. elitists), a crossover probability of 0.7, a mutation probability of 0.3 and a population of 100 individuals. The selection mechanism adopted is the roulette wheel selection and $maxgen_{fitness}$ is set to 100.

The QWS Dataset [2] is used as the QoS dataset of candidate web services. We choose response time, availability, reliability and throughput as the QoS parameters because they are applicable to all web services and are also highly related to the performance of web services. The size of problem represents both the component number in the composite service and the number of candidate services for each component, e.g. x = 20 indicates there are 20 component services in the composite service and 20 candidate services for each one. We set the size of problem from 5 to 45 with the step of 5, and for each case we simulate 10,000 user requests and measure the computing time of both methods to get a concrete composite service. The metric mean computing time (MCT) is calculated as total computing time for all requests divided by number of requests. For more intuitional comparison, we set η as cached GA's MCT divided by GA's MCT.

Since high capacity consumes high memory space and low capacity leads to a low hit ratio, we can learn from Fig. 4 that the capacity of 10000 is a good tradeoff between hit ratio and memory space. Therefore, we set the capacity of the cache pool to 10,000 and use LRU as the replacement policy. We also set p_c to 0.3 and λ to 2, and we focus whether the reduced percentage of MCT in the specific setting can accord with our former study.

The experiments are conducted on a PC Pentium 4 2.8GHz with 2GB RAM, Windows XP SP3, Java 2 Enterprise Edition V1.6.0 and the result is depicted in Fig.5 and Tab.3. As can be seen from Fig 5, with the size of problem increasing, the MCT both approaches consume grow fast and the time used by Cached GA is much less than by GA. The MCT that the cached GA consumes is almost 40% less than the common GA, that is, the caching mechanism reduces almost 40% computing time for GA. The reduced percentage (40%) is nearly equal to the cache hit ratio when the capacity is 10,000, which accords with our study in Section 6.2.

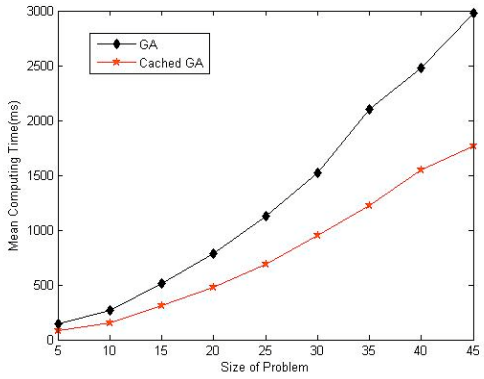

Figure 5 GA vs. Cached GA

Size of Problem	5	10	15	20	25	30	35	40	45
GA's MCT	142	270	516	783	1183	1522	2104	2476	2976
Cached GA's MCT	83	157	310	476	688	950	1227	1550	1765
η	0.585	0.582	0.601	0.608	0.582	0.624	0.583	0.626	0.593

Table 3 Comparison between GA and Cached GA

7 Conclusion and Future work

This paper presented a caching mechanism for QoS-aware service composition, which can be complementary to most of current approaches to enhance the efficiency. We lead an exploratory trial to discretize the continuous QoS constraints and QoS priority weights, and learn the hit ratio of cache under different settings. We then utilize the caching mechanism on genetic algorithm and conduct an experiment on the QWS dataset. Our evaluations show a significant reduction of mean computing time on the problem. This can be especially useful for applications with real-time requirements.

Future work will aim to apply the proposed approach to some large-scale service-oriented system, and to perform a thorough study of user preferences in QoS-aware service composition since some views of preferences in this paper is empirical. We will also focus on the adaptation strategy for the validity time of cache.

Acknowledgments

This work is supported by China's National Science & Technology Pillar Program Project No. 2011BAH25B01 and Fundamental Research Funds for the Central Universities No. CDJXS11181161.

The authors would also like to thank the reviewers for their valuable comments and suggestions.

References

1. E. Al-Masri, and Q. H. Mahmoud, "Investigating web services on the world wide web," in Proceeding of the 17th international conference on World Wide Web (WWW '08). pp. 795-804.
2. E. Al-Masri, and Q. H. Mahmoud, "Qos-based discovery and ranking of web services," in Proceedings of 16th International Conference on Computer Communications and Networks (ICCCN). pp. 529-534.
3. M. Alrifai, and T. Risse, "Combining global optimization with local selection for efficient QoS-aware service composition," in Proceedings of the 19th international conference on World wide web (WWW '09). pp. 881-890.
4. M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for QoS-based web service composition," in Proceedings of the 19th international conference on World wide web (WWW '10). pp. 11-20.
5. R. Berbner, M. Spahn, N. Repp *et al.*, "Heuristics for qos-aware web service composition," in IEEE International Conference on Web Services (ICWS '06). pp. 72-82.
6. G. Canfora, M. Di Penta, R. Esposito *et al.*, "An approach for QoS-aware service composition based on genetic algorithms," in Proceedings of the 2005 conference on Genetic and evolutionary computation (GECCO '05). pp. 1069-1075.
7. J. Cardoso, A. Sheth, J. Miller *et al.*, "Quality of service for workflows and web service processes," Web Semantics: Science, Services and Agents on the World Wide Web, vol. 1, no. 3, pp. 281-308, 2004.
8. Y. Liu, A. H. Ngu, and L. Z. Zeng, "QoS computation and policing in dynamic web service selection," in Proceedings of the 13th International World Wide Web Conference. pp. 66-73.
9. N. Milanovic, and M. Malek, "Current solutions for web service composition," Internet Computing, IEEE, vol. 8, no. 6, pp. 51-59, 2004.
10. OASIS. "Web services business process execution language," http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.
11. M. P. Papazoglou, P. Traverso, S. Dustdar *et al.*, "Service-oriented computing: State of the art and research challenges," Computer, vol. 40, no. 11, pp. 38-45, 2007.

12. M. Stollberg, M. Hepp, and J. Hoffmann, “A caching mechanism for semantic web service discovery,” The Semantic Web, pp. 480-493, 2007.
13. A. Strunk, "QoS-Aware Service Composition: A Survey," in Proceedings of the 2010 Eighth IEEE European Conference on Web Services (ECOWS '10). pp. 67-74.
14. X. Ye, and R. Mounla, "A hybrid approach to QoS-aware service composition," in IEEE International Conference on Web Services (ICWS '08). pp. 62-69.
15. T. Yu, and K. J. Lin, "Service selection algorithms for composing complex services with multiple QoS constraints," in Service-Oriented Computing-ICSOC. pp. 130-143.
16. M. Zeleny, Multiple criteria decision making: McGraw-Hill New York, 1982.
17. L. Z. Zeng, B. Benatallah, A. H. H. Ngu *et al.*, “QoS-aware middleware for Web Services Composition,” IEEE Transactions on Software Engineering, vol. 30, no. 5, pp. 311-327, May, 2004.
18. Z. Zheng, H. Ma, M. R. Lyu *et al.*, "Wsrec: A collaborative filtering based web service recommender system," in IEEE International Conference on Web Services (ICWS '09). pp. 437-444.